2d-transform

A program by Way Yan

Introduction

`2d-transform` was developed while I was learning multivariable calculus. It is used to visualise functions f from $\mathbb{R}^2$ to $\mathbb{R}^2$. It does so by applying f to the gridlines in the domain, then drawing the result as a pdf file. I use pdf instead of other file formats like png and svg, because pdfs can be scaled without losing quality unlike pngs, and they are natively supported by `pdftex` unlike svgs.

Usage

There should be two shell scripts in the same directory as this file, `compile.sh` and `build.sh`. `build.sh` generates the LaTeX documentation and C source from `2d-transform.nw` and additionally compiles the C source into the executable `2d-transform`. `compile.sh` only does the second step.

Initially, `2d-transform` will generate an output pdf with a set of default parameters. To run `2d-transform` with different parameters, the user first has to edit the ⟨*parameters*⟩ section in `2d-transform.c`. Then, the user should run `compile.sh` to generate a new executable. This executable, when run, produces a pdf file based on the parameters specified earlier.

The main reason for requiring the parameters to be specified in the source code is to make my life as a programmer easier. If the function f were to be specified instead as a command-line argument to the executable, I would have to add code to parse the definition of f as a mathematical expression. Though it is certainly not impossible to implement, I find it an unwanted distraction.

The following code outlines the top-level structure of `2d-transform`:

```
⟨2d-transform.c⟩≡
  ⟨headers⟩
  ⟨parameters⟩
  ⟨internal state⟩
  ⟨cartesian-to-cairo-coords⟩
  int main() {
          ⟨setup⟩
          ⟨draw⟩
          ⟨cleanup⟩
  }
```

Headers

2d-transform has very few dependencies, hence this section is small. stdio consists of standard input/output functions (mainly printf for debugging), math allows the user to input functions like sin in ⟨*parameters*⟩, and cairo is the graphics library responsible for drawing. I use cairo primarily because it supports pdf output, unlike other libraries.

⟨*headers*⟩≡

```
#include <stdio.h>
#include <math.h>
#include <cairo.h>
```

Parameters

These are the things that the user has to specify before compiling 2d-transform.c and running the executable to obtain the pdf.

Of course, the main thing we want to specify is the actual function to draw. These are provided through the parameters f1 and f2, which represent the x and y components of f. If the source code is not modified, then the default function is the identity.

⟨*parameters*⟩≡

```
#define f1(x,y) sin(x)
#define f2(x,y) y+sin(x)
⟨gridline parameters⟩
⟨test square parameters⟩
⟨misc parameters⟩
```

Next are the parameters for the drawing of gridlines. Essentially, the domain is defined to be

$$[\texttt{dom\_xmin}, \texttt{dom\_xmax}] \times [\texttt{dom\_ymin}, \texttt{dom\_ymax}].$$

Only the portions of the gridlines lying in the domain are considered for the drawing. Similarly, the codomain is defined to be

$$[\texttt{codom\_xmin}, \texttt{codom\_xmax}] \times [\texttt{codom\_ymin}, \texttt{codom\_ymax}].$$

If you imagine the codomain to be a rectangle in $\mathbb{R}^2$ containing the transformed gridlines from the domain, then the pdf output will look like that rectangle, but scaled to the dimension specified by `pdfsize`. Also, `gridlinegap` specifies the distance between consecutive horizontal and vertical gridlines in the domain.

Now, each gridline L has infinitely many points—hence one cannot hope to fully determine $f(L)$ by computing $f(x)$ for every single point $x \in L$. Rather, in the case where L is a vertical line, the program will go from top to bottom, sampling a finite number of uniformly spaced points x_i along L (see ⟨*draw one vertical gridline*⟩ for more details). The spacing between the points x_i is governed by `samplegap`. The smaller `samplegap` is, the more accurate the output will be. Both `gridlinegap` and `samplegap` have no units, because they represent lengths in the coordinate system of the domain.

⟨*gridline parameters*⟩≡

```
#define dom_xmin -5
#define dom_xmax 5
#define dom_ymin -5
#define dom_ymax 5
#define codom_xmin -5
#define codom_xmax 5
#define codom_ymin -5
#define codom_ymax 5
#define gridlinegap 0.5
#define samplegap 0.1
```

`2d-transform` also has a feature to visualise the image of a specified square under f. It is enabled by setting `tsenable` to 1 and disabled by setting it to 0. The difference between this and gridlines is that the area of the square is filled in with points, and also the sides and corners of the square are drawn in such a way that they can be distinguished. Hence this feature reveals some features of f that are otherwise hard to tell, such as orientation. It is also helpful when there are many gridlines mapping to the same region in the domain.

The square, as a subset of the domain, is defined as

$$[\texttt{ts\_xmin}, \texttt{ts\_xmax}] \times [\texttt{ts\_ymin}, \texttt{ts\_ymax}].$$

Also, `tsgap` specifies the gap to leave between points when filling up the area of the square. As with `gridlinegap` and `samplegap`, the parameter `tsgap` has no units because it is relative to the coordinate system of the codomain.

⟨*test square parameters*⟩≡
```
#define tsenable 1
#define ts_xmin -2
#define ts_xmax 2
#define ts_ymin -2
#define ts_ymax 2
#define tsgap 0.4
```

The original and transformed test square is shown in Figures 1 and 2. Note that as one goes from left to the right, the size of the circles transitions from `ts_startsize` to `ts_endsize`. Likewise as one goes from bottom to top, the colour transitions from (`ts_startr`, `ts_startg`, `ts_startb`) to (`ts_endr`, `ts_endg`, `ts_endb`).

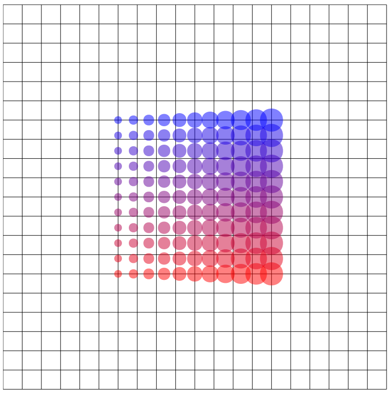

Figure 1: How the square looks like before transformation.

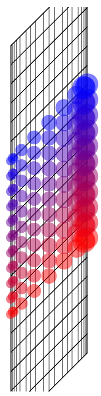

Figure 2: How the square looks like after the transformation $(\sin x,\ y + \sin x)$.

⟨*test square parameters*⟩+≡

```
#define ts_startsize 1
#define ts_endsize 3
#define ts_startr 255
#define ts_startg 0
#define ts_startb 0
```

```
#define ts_endr 0
#define ts_endg 0
#define ts_endb 255
```

Lastly we have miscellaneous parameters. `pdfsize` is width of the pdf, in points. (For reference, an A4-sized pdf is 595 points wide and 842 points high.) Actually, `pdfsize` is also the height of the pdf, because I make the width and height equal for simplicity. `outputfile` is the name of the pdf file to save the drawing to.

⟨*misc parameters*⟩≡

```
#define pdfsize 100
#define outputfile "out.pdf"
```

Internal state

These are variables that are used internally, and should be not be touched by the user. Firstly, I provide the function definition of `cairo_pdf_surface_create`; otherwise, the compiler throws an `Implicit function declaration` error (I'm not fully sure why). Then, `surface` and `cr` is used by `cairo` for drawing.

Lastly, `pair` is effectively a register variable that stores the output after calling `cartesian_to_cairo_coords`. It is implemented this way, because C does not allow `cartesian_to_cairo_coords` to return an array directly. It can return a pointer to an array, but I find it completely unnecessary to deal with pointers for such a small array. Hence I resorted to the simplest solution.

⟨*internal state*⟩≡

```
cairo_surface_t *cairo_pdf_surface_create(const char *filename,
        double width_in_points, double height_in_points);
cairo_surface_t *surface;
cairo_t *cr;
double pair[2];
```

Changing coordinate systems

The coordinate system that `cairo` uses is different from the Cartesian coordinate system: the top-left corner is `(0,0)` while the bottom-right corner is `(pdfsize,pdfsize)`. In Cartesian coordinates, `(0,0)` would be at the center.

The function takes in the Cartesian coordinates (x, y) as input. It computes the values $t_x, t_y \in [0, 1]$ such that

$$x = (1 - t_x) * \texttt{codom\_xmin} + t_x * \texttt{codom\_xmax} \quad \text{and}$$
$$y = (1 - t_y) * \texttt{codom\_ymax} + t_y * \texttt{codom\_ymin}.$$

Essentially, t_x measures how far right (x, y) should be in the pdf file from the left; t_y measures how far down it should be from the top. Note that going up in a pdf file corresponds to decreasing the y coordinate. Then, we simply scale t_x and t_y by `pdfsize`.

⟨*cartesian-to-cairo-coords*⟩≡
```
void cartesian_to_cairo_coords(double x, double y) {
        double tx = (x-codom_xmin)/(codom_xmax-codom_xmin);
        double ty = (y-codom_ymax)/(codom_ymin-codom_ymax);
        pair[0] = tx * pdfsize;
        pair[1] = ty * pdfsize;
}
```

Setup and cleanup

Nothing much to say really.

⟨*setup*⟩≡
```
surface = cairo_pdf_surface_create(outputfile, pdfsize, pdfsize);
cr = cairo_create(surface);
cairo_set_source_rgb(cr, 0, 0, 0);
cairo_set_line_width(cr, 0.1);
```

⟨*cleanup*⟩≡
```
cairo_destroy(cr);
cairo_surface_destroy(surface);
return 0;
```

Drawing

This is the real meat of the program.

⟨*draw*⟩≡

```
double xcur, ycur;
⟨draw transformed vertical gridlines⟩
⟨draw transformed horizontal gridlines⟩
if (tsenable) {
        ⟨draw test square⟩
}
```

⟨*draw transformed vertical gridlines*⟩≡

```
⟨find leftmost vertical gridline⟩
⟨iterate through vertical gridlines⟩
```

Besides being equally spaced according to `gridlinegap`, I require that one of the vertical gridlines be the y-axis. Doing so fixes all the other vertical gridlines in $\mathbb{R}^2$ and hence D.

With this, the leftmost vertical gridline is the `n`th one from the y-axis. If `n>0`, then it lies to the right of the y-axis. If `n<0` then it lies to the left.

⟨*find leftmost vertical gridline*⟩≡

```
int n = dom_xmin / gridlinegap;
xcur = n * gridlinegap;
```

The vertical gridlines which intersect D are drawn.

⟨*iterate through vertical gridlines*⟩≡

```
while (xcur <= dom_xmax) {
        ⟨draw one vertical gridline⟩
        cairo_stroke(cr);
        xcur += gridlinegap;
}
```

The point (xcur,ycur) iterates through the sampled points x_i (see ⟨*parameters*⟩). Lines are drawn connecting each pair $(f(x_i), f(x_{i+1}))$, the idea being that the resulting collection of line segments closely resembles $f(L)$.

⟨*draw one vertical gridline*⟩≡

```
ycur = dom_ymin;
cartesian_to_cairo_coords(f1(xcur,ycur), f2(xcur,ycur));
cairo_move_to(cr, pair[0], pair[1]);
while (ycur <= dom_ymax) {
        cartesian_to_cairo_coords(f1(xcur,ycur), f2(xcur,ycur));
        cairo_line_to(cr, pair[0], pair[1]);
        ycur += samplegap;
}
```

The code for drawing horizontal gridlines is essentially the same except the characters x and y are swapped. Hence I will not divide it further.

⟨*draw transformed horizontal gridlines*⟩≡

```
n = dom_ymin / gridlinegap;
ycur = n * gridlinegap;
while (ycur <= dom_ymax) {
        xcur = dom_xmin;
        cartesian_to_cairo_coords(f1(xcur,ycur), f2(xcur,ycur));
        cairo_move_to(cr, pair[0], pair[1]);
        while (xcur <= dom_xmax) {
                cartesian_to_cairo_coords(f1(xcur,ycur), f2(xcur,ycur));
                cairo_line_to(cr, pair[0], pair[1]);
                xcur += samplegap;
        }
        cairo_stroke(cr);
        ycur += gridlinegap;
}
```

Next, the test square is drawn.

⟨*draw test square*⟩≡

```
double curx, cury;
double tx, ty, r, g, b, size;
curx = ts_xmin;
while (curx <= ts_xmax) {
        cury = ts_ymin;
        while (cury <= ts_ymax) {
                ⟨draw circle⟩
                cury += tsgap;
        }
        curx += tsgap;
}
```

⟨*draw circle*⟩≡

```
tx = (curx-ts_xmin)/(ts_xmax-ts_xmin);
ty = (cury-ts_ymin)/(ts_ymax-ts_ymin);
r = ((1-ty)*ts_startr + ty*ts_endr) / 255;
g = ((1-ty)*ts_startg + ty*ts_endg) / 255;
b = ((1-ty)*ts_startb + ty*ts_endb) / 255;
size = (1-tx)*ts_startsize + tx*ts_endsize;
cairo_set_source_rgba(cr, r, g, b, 0.5);
cartesian_to_cairo_coords(f1(curx,cury), f2(curx, cury));
cairo_arc(cr, pair[0], pair[1], size, 0, 2*M_PI);
cairo_fill(cr);
```